## JavaScript Card Objects

This article shows how user-defined JavaScript objects can be implemented and used in a web page. Part of the example here also makes use of CSS and DHTML but the main focus is on creating and using objects in JavaScript.

### Overview

The assumption is that our objects will be used to implement some card games in JavaScript. To provide basic functionality we'll define two objects, Card and Stack.

The Card object will be used to represent individual playing cards while the Stack object will represent a set of cards, which individual cards can be added to or removed from. The Stack object will be designed so it can act as a full deck of cards or as smaller sets of cards like individual player hands or a discard pile.

#### Ranks and Suits

A standard pack of cards contains thirteen cards of different ranks (ace, 2, 3, ... 10, jack, queen and king) in each of four suits (clubs, diamonds, hearts and spades).

To represent these values, a one or two character string will be used. For rank, the values are "A", "2", "3", ..., "10", "J", "Q" and "K" and for suits "C", "D", "H" and "S" are used. These values are seen throughout the code for the objects and can likewise be used in any script that needs to evaluate a card's value.

### The Card Object

The first JavaScript object we'll create is the Card object. Below is the constructor function.

function Card(rank, suit) {

this.rank = rank;

this.suit = suit;

this.toString = cardToString;

this.createNode = cardCreateNode;

}

As you can see, the card's rank and suit are saved as properties of the same name. Two methods are defined as well, toString() which will return the value of the card as a text string and createNode() which can be used for graphically display the card. This latter method is discussed in detail later on.

The toString() method is fairly simple to implement. The code is shown below.

function cardToString() {

var rank, suit;

switch (this.rank) {

case "A" :

rank = "Ace";

break;

case "2" :

rank = "Two";

break;

case "3" :

rank = "Three";

break;

case "4" :

rank = "Four";

break;

case "5" :

rank = "Five";

break;

case "6" :

rank = "Six";

break;

case "7" :

rank = "Seven";

break;

case "8" :

rank = "Eight";

break;

case "9" :

rank = "Nine";

break;

case "10" :

rank = "Ten";

break;

case "J" :

rank = "Jack"

break;

case "Q" :

rank = "Queen"

break;

case "K" :

rank = "King"

break;

default :

rank = null;

break;

}

switch (this.suit) {

case "C" :

suit = "Clubs";

break;

case "D" :

suit = "Diamonds"

break;

case "H" :

suit = "Hearts"

break;

case "S" :

suit = "Spades"

break;

default :

suit = null;

break;

}

if (rank == null || suit == null)

return "";

return rank + " of " + suit;

}

Given the above, if you used this code,

var myCard = new Card("3", "C");

alert(myCard.toString());

the browser would display "Three of Clubs." In other words, it just spells out the rank and suit of the card. If either rank or suit are invalid, a null string is returned ("").

It's interesting to note all JavaScript objects have a default toString() method which is used internally for things like error messages or anytime an object is used in a string expression. Because of this you could also use just the variable name in the alert()call:

alert(myCard);

Most built-in JavaScript objects have their own implementation of toString() which returns a suitable value for the object or data type the variable represents. For other objects it usually returns a string value in the form "[object]" or "[object *type*]." Defining our own toString() method let's us give a string value with more useful information.

**The Stack Object**

See the card stacks[**demo**](http://www.brainjar.com/js/cards/stacksdemo.html).

The Stack object is designed to make it easier for a programmer to use Card objects in a manner similar to how a real deck or stack of cards is used. Basically, it creates an ordered set of cards that can be drawn from, added to, shuffled or combined with other Stacks.

The constructor for the Stack object is shown below.

function Stack() {

// Create an empty array of cards.

this.cards = new Array();

this.makeDeck = stackMakeDeck;

this.shuffle = stackShuffle;

this.deal = stackDeal;

this.draw = stackDraw;

this.addCard = stackAddCard;

this.combine = stackCombine;

this.cardCount = stackCardCount;

}

There's not much to it other than creating an empty array for the cards and assigning the methods. Many of these methods will in fact make use of the methods already provided by the built-in JavaScript Array object to manipulate the set of cards.

**Building a Deck of Cards**

The first method, makeDeck(), is provided save a user of the object some work by generating a full deck of cards in the stack.

function stackMakeDeck(n) {

var ranks = new Array("A", "2", "3", "4", "5", "6", "7", "8", "9",

"10", "J", "Q", "K");

var suits = new Array("C", "D", "H", "S");

var i, j, k;

var m;

m = ranks.length \* suits.length;

// Set array of cards.

this.cards = new Array(n \* m);

// Fill the array with 'n' packs of cards.

for (i = 0; i < n; i++)

for (j = 0; j < suits.length; j++)

for (k = 0; k < ranks.length; k++)

this.cards[i \* m + j \* ranks.length + k] =

new Card(ranks[k], suits[j]);

}

Note that it takes an integer argument. Some card games may require you to combine two or more regular packs of cards into a single deck for play. This argument let's you specify how many packs to include in the deck.

The first step is to recreate the array of Card objects based on the number of cards needed. Then it loops through the rank and suit values, creating a Card object for each and placing it in the cards array. The process is repeated n times.

This creates a nicely ordered set of cards. Of course, the first thing you'll probably want to do with a new deck is destroy that nice order by shuffling them.

**Shuffling a Stack**

The shuffle() method accomplishes this by randomizing the order of the cards currently in the stack.

function stackShuffle(n) {

var i, j, k;

var temp;

// Shuffle the stack 'n' times.

for (i = 0; i < n; i++)

for (j = 0; j < this.cards.length; j++) {

k = Math.floor(Math.random() \* this.cards.length);

temp = this.cards[j];

this.cards[j] = this.cards[k];

this.cards[k] = temp;

}

}

It takes a numeric argument which defines how many times the stack should be shuffled. For each shuffle it simply loops though every card in the array and swaps it with another card randomly selected from the array.

**Dealing Cards from a Stack**

The deal() method simulates dealing a single card from the stack. The first card is removed from the stack and returned to the caller.

function stackDeal() {

if (this.cards.length > 0)

return this.cards.shift();

else

return null;

}

The draw() method does the same thing, except that it allows you to specify any card in the stack by passing an index value in the call.

function stackDraw(n) {

var card;

if (n >= 0 && n < this.cards.length) {

card = this.cards[n];

this.cards.splice(n, 1);

}

else

card = null;

return card;

}

The index value is zero-based, so draw(0) is equivalent to deal(). Note that if no cards are left in a stack or if an invalid value is given, null is returned. You can find out how many cards are in a stack at any given time using the cardCount() method.

function stackCardCount() {

return this.cards.length;

}

**Adding Cards to a Stack**

The addCard() method appends a Card object to the end of the stack's card array.

function stackAddCard(card) {

this.cards.push(card);

}

As and example, you could define one Stack for a deck and one for a poker hand, then deal from the deck to the hand.

var deck = new Stack();

var hand = new Stack();

var i;

deck.makeDeck(1);

for (i = 0; i < 5; i++)

hand.addCard(deck.deal());

**Combining Stacks**

The final method let's you combine two stacks.

function stackCombine(stack) {

this.cards = this.cards.concat(stack.cards);

stack.cards = new Array();

}

It takes a Stack object as an argument and appends its cards to the current stack. Note that the stack being appended is emptied of cards. So if stack1 contained A♣, 7♥, 3♦, Q♥ and stack2 contained K♠, 5♣, the call

stack1.combine(stack2);

would leave stack1 with A♣, 7♥, 3♦, Q♥, K♠, 5♣ while stack2 would be empty.

With the Stack object in place, you can play around with a [**demo**](http://www.brainjar.com/js/cards/stacksdemo.html) to see how they might be put to use in managing sets of cards.

**Displaying Cards**

See [**CSS Playing Cards**](http://www.brainjar.com/css/cards/)and [**Introduction to the Document Object Model**](http://www.brainjar.com/dhtml/intro/)for  details.

As mentioned previously, the Card object'screateNode() method can be used to create a graphical display of a card that can be dynamically added to a page.

This requires some style rules to be defined and the use of the Document Object Model (DOM) to create the necessary elements and insert them in the page. Details on the CSS and coding techniques are available elsewhere on this site but we'll go over the basics here.

**The createNode() Method**

This method generates a DOM Node object consisting of various HTML elements with style class assignments and graphics based on the card's rank and suit.

function cardCreateNode() {

var cardNode, frontNode, indexNode, spotNode, tempNode, textNode;

var indexStr, spotChar;

// This is the main node, a DIV tag.

cardNode = document.createElement("DIV");

cardNode.className = "card";

// Build the front of card.

frontNode = document.createElement("DIV");

frontNode.className = "front";

// Get proper character for card suit and change font color if

// necessary.

spotChar = "\u00a0";

switch (this.suit) {

case "C" :

spotChar = "\u2663";

break;

case "D" :

frontNode.className += " red";

spotChar = "\u2666";

break;

case "H" :

frontNode.className += " red";

spotChar = "\u2665";

break;

case "S" :

spotChar = "\u2660";

break;

}

// Create and add the index (rank) to the upper-left corner of the

// card.

indexStr = this.rank;

if (this.toString() == "")

indexStr = "\u00a0";

spotNode = document.createElement("DIV");

spotNode.className = "index";

textNode = document.createTextNode(indexStr)

spotNode.appendChild(textNode);

spotNode.appendChild(document.createElement("BR"));

textNode = document.createTextNode(spotChar)

spotNode.appendChild(textNode);

frontNode.appendChild(spotNode);

// Create and add spots based on card rank (Ace thru 10).

spotNode = document.createElement("DIV");

textNode = document.createTextNode(spotChar);

spotNode.appendChild(textNode);

if (this.rank == "A") {

spotNode.className = "ace";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "3" || this.rank == "5" || this.rank == "9") {

spotNode.className = "spotB3";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "2" || this.rank == "3") {

spotNode.className = "spotB1";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "2" || this.rank == "3") {

spotNode.className = "spotB5";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "4" || this.rank == "5" || this.rank == "6" ||

this.rank == "7" || this.rank == "8" || this.rank == "9" ||

this.rank == "10") {

spotNode.className = "spotA1";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotA5";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC1";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC5";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "6" || this.rank == "7" || this.rank == "8") {

spotNode.className = "spotA3";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC3";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "7" || this.rank == "8" || this.rank == "10") {

spotNode.className = "spotB2";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "8" || this.rank == "10") {

spotNode.className = "spotB4";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

if (this.rank == "9" || this.rank == "10") {

spotNode.className = "spotA2";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotA4";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC2";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC4";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

// For face cards (Jack, Queen or King), create and add the proper

// image.

tempNode = document.createElement("IMG");

tempNode.className = "face";

if (this.rank == "J")

tempNode.src = "graphics/jack.gif";

if (this.rank == "Q")

tempNode.src = "graphics/queen.gif";

if (this.rank == "K")

tempNode.src = "graphics/king.gif";

// For face cards, add suit characters to the upper-left and

// lower-right corners.

if (this.rank == "J" || this.rank == "Q" || this.rank == "K") {

frontNode.appendChild(tempNode);

spotNode.className = "spotA1";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

spotNode.className = "spotC5";

tempNode = spotNode.cloneNode(true);

frontNode.appendChild(tempNode);

}

// Add front node to the card node.

cardNode.appendChild(frontNode);

// Return the card node.

return cardNode;

}

Note the character strings "\u00a0," "\u2663," "\u2666," "\u2665," and "\u2660." These are Unicode escape sequences for the characters corresponding to the HTML character entities &nbsp;, &clubs;, &diams;, &hearts; and &spades; respectively.

The node returned corresponds to a string of HTML. Combined with the proper CSS rules it will render as a fairly decent graphic representation of a playing card. For example, the HTML for the 3♠ card would look like this:

<div class="card">

<div class="front">

<div class="index">3<br />&spades;</div>

<div class="spotB1">&spades;</div>

<div class="spotB3">&spades;</div>

<div class="spotB5">&spades;</div>

</div>

</div>

and appear on the page like this.

**3  
♠**

♠

♠

♠

The node returned by the method reflects the outer DIV tag. This node can be used just like any other DIV element node in a page. Should the card have an invalid rank or suit value, the node returned will appear as a blank-faced card.

**Images**

Some images are required for face cards and the card back. To help speed things up, the following code is included to preload the necessary graphics.

var cardImg0 = new Image(); cardImg0.src= "graphics/cardback.gif";

var cardImg1 = new Image(); cardImg1.src= "graphics/jack.gif";

var cardImg2 = new Image(); cardImg2.src= "graphics/queen.gif";

var cardImg3 = new Image(); cardImg3.src= "graphics/king.gif";

**Using Card Nodes**

These nodes can be added to a page using any of several DOM methods. For example:

var deck, card;

deck = new Stack();

deck.makeDeck(1);

card = deck.deal();

document.appendChild(card.createNode());

Note that appendChild(), like similar DOM methods, inserts a copy of the given node into the document and returns a reference to that copy. In order to access the card node once it's been added to the page you need to save this reference. Then you can easily locate and remove it if desired.

var node;

...

// Add to page.

node = document.appendChild(card.createNode());

...

// Remove from page.

document.removeChild(node);

**Face Down Cards**

The outer DIV of the card node uses a graphic background to represent the back of a playing card. This is normally overlaid by the inner DIV which represents the card's front. But you can access this DIV element from the card node and change its visibilitystyle to "flip" the card over and  make it appear to be face down.

node = document.appendChild(card.createNode());

...

// Face down.

node.firstChild.style.visibility = "hidden";

...

// Face up.

node.firstChild.style.visibility = "";

The display [**demo**](http://www.brainjar.com/js/cards/displaydemo.html) illustrates some of these techniques for working with card nodes using the same basic concept as the previous demo.

**Source**

The JavaScript code, style definitions and images can be viewed and copied here. See the demos for examples of use.

* [**cards.css**](http://www.brainjar.com/common/viewsource.asp?source=\js\cards\cards.css) - Style definitions.
* [**cards.js**](http://www.brainjar.com/common/viewsource.asp?source=\js\cards\cards.js) - JavaScript code.
* [**cardback.gif**](http://www.brainjar.com/js/cards/graphics/cardback.gif)
* [**jack.gif**](http://www.brainjar.com/js/cards/graphics/jack.gif)
* [**queen.gif**](http://www.brainjar.com/js/cards/graphics/queen.gif)
* [**king.gif**](http://www.brainjar.com/js/cards/graphics/king.gif)

**Conclusion**

By defining your own objects in JavaScript you can create abstract data types to represent just about anything. This allows you to simplify your program designs by encapsulating complex code into simple properties and method calls. Objects are also easier to reuse, modify or expand on.

For fun you can try this [**Blackjack**](http://www.brainjar.com/js/blackjack/blackjack.html) game, made using the objects discussed here. Details on the coding behind it can be found in a separate [**article**](http://www.brainjar.com/js/blackjack/).